NumPy:

NumPy stands for 'Numerical Python' or 'Numeric Python'. It is an open source module of Python which provides fast mathematical computation on arrays and matrices. Since, arrays and matrices are an essential part of the Machine Learning ecosystem, NumPy along with Machine Learning modules like Scikit-learn, Pandas, Matplotlib, TensorFlow, etc. complete the Python Machine Learning Ecosystem.

NumPy is extremely **important for Data Science** because:

- Linear algebra library
- Powerful and incredibly fast
- Integrate C/C++ and Fortran code

Almost all of the PyData (https://pydata.org) Eco-System libraries rely on NumPy (http://www.numpy.org). This is one of their **most important and main building block**. In this section, we will cover the key concepts of this wonderful Python library.

To use numpy, you need to import the module:

In [110]:

```
import numpy as np
import time
import sys
import matplotlib.pyplot as plt
```

Creating Numpy arrays

There are a number of ways to initialize new Numpy arrays, for example from

- From Python data type (e.g. List, Tuple)
- using functions that are dedicated to generating numpy arrays, such as `arange` , `linspace` , etc.
- reading data from files

From lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array` function

In [2]:

```
# a vector: the argument to the array function is a Python list
v = np.array([1,2,3,4])
v
```

Out[2]:

```
array([1, 2, 3, 4])
```

In [3]:
```
# a matrix: the argument to the array function is a nested Python list
M = np.array([[1, 2], [3, 4]])
M
```

Out[3]:
```
array([[1, 2],
       [3, 4]])
```

So far the `numpy.ndarray` looks a lot like a Python list (or nested list). Why not simply use Python lists for computations instead of creating a new array type?

Advantages of using NumPy Arrays:

The most important benefits of using it are :

- It consumes less memory.
- It is fast as compared to the python List.
- It is convenient to use.

Program Showing Memory-- Numpy Vs List

In [4]:
```
# Creating a NumPy array with 100 elements
array = np.arange(100)
# array.itemsize : Size of one element
# array.size : length of array
print("Size of NumPy array: ", array.size * array.itemsize)

# Creating a list with 100 elements
# Now I'll print the size of list
list = range(0, 100)
# Multiplying size of 1 element with length of the list
print("Size of list: ", sys.getsizeof(1)*len(list))
```

```
Size of NumPy array:  400
Size of list:  2800
```

Program Showing Execution Time and convenience-- Numpy Vs List

In [5]:

```python
# Let's declare the size
Size = 100000

# Creating two lists
list1 = range(Size)
list2 = range(Size)

# Creating two NumPy arrays
arr1 = np.arange(Size)
arr2 = np.arange(Size)

# # Calculating time for Python List
start = time.time()
result = [(x+y) for x, y in zip(list1, list2)]

print("Time for Python List in msec: ", (time.time() - start) * 1000)

# # Calculating time for NumPy array
start = time.time()
result = arr1+arr2
print("Time for NumPy array in msec: ", (time.time()- start) * 1000)

print("\nThis means NumPy array is faster than Python List")
```

```
Time for Python List in msec:  15.623807907104492
Time for NumPy array in msec:  0.0

This means NumPy array is faster than Python List
```

Using array-generating functions

For larger arrays it is inpractical to initialize the data manually, using explicit pythons lists. Instead we can use one of the many functions in `numpy` that generates arrays of different forms.

Most of the times, we use NumPy built-in methods to create arrays. These are much simpler and faster.

Some of the more common are:

Cat1
- rand
- random or random_sample

Cat2.
- randn
- normal

- arange()
- linspace()
- zeros()
- ones()
- eye()
- diag()
- full()
- **Random**
  - rand()
  - random()
  - randn()
  - normal()
  - randint()

Diff: Among the functions belong Cat 1

① both give o/p for uniform distributions, values in between [0, 1)

② looks similar, when no argument supplied.

③ However they are different, when we pass the arguments.

rand → Each dimensions as separated argument not as tuple.

random → Here arguments for dimensions are passed as tuple.

Diff: Among category 2

randn	normal
① Tuned for Std. Normal distn. ie ($\mu=0, \sigma=1$)	① Generalized Normal distn ie μ & σ values can be controlled. By default give o/p with $\mu=0, \sigma=1$
② Need to pass each dim- as separated elements	② Here, dimensions arguments are passed as tuple.

    - choice()
- reshape()

a. `arange()`

- arange() is very much similar to Python function range()
- Syntax: arange([start,] stop[, step,], dtype=None)
- Return evenly spaced values within a given interval.

In [6]:
```
# create a range (the end value is not included)
x = np.arange(-1, 1, 0.1) # arguments: start, stop, step
x
```

Out[6]:
```
array([-1.00000000e+00, -9.00000000e-01, -8.00000000e-01, -7.00000000e-01,
       -6.00000000e-01, -5.00000000e-01, -4.00000000e-01, -3.00000000e-01,
       -2.00000000e-01, -1.00000000e-01, -2.22044605e-16,  1.00000000e-01,
        2.00000000e-01,  3.00000000e-01,  4.00000000e-01,  5.00000000e-01,
        6.00000000e-01,  7.00000000e-01,  8.00000000e-01,  9.00000000e-01])
```

In [7]:
```
# range of integers
y = np.arange(0, 10, 1) # arguments: start, stop, step
y
```

Out[7]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [8]:
```
# specifying dtype as float
z = np.arange(0, 10, 1, dtype=float) # arguments: start, stop, step
z
```

Out[8]:
```
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

b. `linspace()`

Return evenly spaced numbers over a specified interval.
Press shift+tab for the documentation.

In [9]:
```
# start from 1 & end at 15 with 15 evenly spaced points b/w 1 to 15.
print(np.linspace(1, 15, 15))
type(np.linspace(1, 15, 15))
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15.]
```

Out[9]:
```
numpy.ndarray
```

In [10]:
```
# Lets find the step size with "retstep" which returns the array and the ste
my_linspace = np.linspace(5, 15, 9, retstep=True)
my_linspace[1]
```

Out[10]:
```
1.25
```

In [11]:
```python
# my_linspace[1] to get the stepsize only
type(my_linspace)
```

Out[11]: tuple

In [12]:
```python
my_linspace[0]
```

Out[12]:
```
array([ 5.  ,  6.25,  7.5 ,  8.75, 10.  , 11.25, 12.5 , 13.75, 15.  ])
```

In [13]:
```python
my_linspace
```

Out[13]:
```
(array([ 5.  ,  6.25,  7.5 ,  8.75, 10.  , 11.25, 12.5 , 13.75, 15.  ]), 1.25)
```

Don't Confuse!

- **arange() takes 3rd argument as step size.**
- **linspace() take 3rd argument as no of point we want.**

c. `zeros()`

- We want to create an array with **all zeros**

Press shift+tab for the documentation.

In [14]:
```python
np.zeros(3,dtype=int) # 1-D with 3 elements
```

Out[14]:
```
array([0, 0, 0])
```

In [15]:
```python
np.zeros((4,6)) #(no_row, no_col) passing a tuple
```

Out[15]:
```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

d. `ones()`

- We want to create an array with **all ones**

Press shift+tab for the documentation.

In [16]:
```python
np.ones(3)
```

Out[16]:
```
array([1., 1., 1.])
```

In [17]:
```
np.ones((4,6)) #(no_row, no_col) passing a tuple
```

Out[17]:
```
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

e. eye()

Creates an identity matrix must be a square matrix, which is useful in several linear algebra problems.

- Return a 2-D array with **ones on the diagonal and zeros elsewhere.**

Press shift+tab for the documentation.

In [18]:
```
np.eye(5)
```

Out[18]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [19]:
```
np.eye(5,5) #(no_row, no_col) 2-D
# not work with tuple, list
```

Out[19]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

f. diag()

In [20]:
```
# diagonal matrix
np.diag([100,2,3])
```

Out[20]:
```
array([[100,   0,   0],
       [  0,   2,   0],
       [  0,   0,   3]])
```

g. full()

In [21]:
```
np.full((3,3),'hello')
```

Out[21]:
```
array([['hello', 'hello', 'hello'],
       ['hello', 'hello', 'hello'],
       ['hello', 'hello', 'hello']], dtype='<U5')
```

Random

We can also create arrays with random numbers using Numpy's built-in functions in Random module.
np.random. and then press tab for the options with random

h. rand()

Create an array of the given shape and populate it with random samples from a uniform continuous distribution over half-open interval `[0, 1)` .

In [22]:
```
np.random.rand(3) # 1-D array with three elements
```

Out[22]:
```
array([0.25291713, 0.13014499, 0.11125678])
```

In [23]:
```
np.random.rand(3,2) # row, col, note we are not passing a tuple here, each d
```

Out[23]:
```
array([[0.82982611, 0.55919205],
       [0.62487402, 0.54381626],
       [0.35722338, 0.34350427]])
```

In [24]:
```
np.random.rand([3,2])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-509995c2e821> in <module>
----> 1 np.random.rand([3,2])

mtrand.pyx in mtrand.RandomState.rand()

mtrand.pyx in mtrand.RandomState.random_sample()

mtrand.pyx in mtrand.cont0_array()

TypeError: 'list' object cannot be interpreted as an integer
```

i. random()

This will return random floats in the half-open interval `[0, 1)` following the “continuous uniform” distribution.

np.random.random((4,3))

In [25]:
```
np.random.random((4,3))    ### OR    np.random.random_sample()----- Both are
```

Out[25]:
```
array([[0.08754939, 0.82112871, 0.36491524],
       [0.02621599, 0.0199023 , 0.45754401],
       [0.39808594, 0.34582793, 0.29034372],
       [0.60447659, 0.39296346, 0.45286649]])
```

Difference between "rand" Vs "random"

"The only difference is in how the arguments are handled. With numpy.random.rand(), the length of each dimension of the output array is a separate argument. With numpy.random.random(), the shape argument is a single tuple

j. randn()

Return a sample (or samples) from the "standard normal" or a "Gaussian" distribution. Unlike rand which is uniform.it gives a distribution from some standardized normal distribution (mean 0 and variance 1).
Press shift+tab for the documentation.

In [26]:
```
np.random.randn(2)
```

Out[26]:
```
array([-2.14416852, -0.7420775 ])
```

In [27]:
```
np.random.randn(7,7) # no tuple, each dimension as a separate argument
```

Out[27]:
```
array([[ 1.2778984 , -1.52179034, -0.27023177, -1.0951708 ,  0.27828555,
        -0.94880213, -0.15412756],
       [ 0.32057002, -0.13943759, -0.45591495, -1.48885516, -1.56800077,
         0.16749054,  1.77579938],
       [-1.13139007,  0.50770725, -0.1104798 ,  0.73944348,  0.11128532,
         0.24428251, -0.84730749],
       [ 1.06311259, -1.92059439, -0.55941951,  1.6224381 ,  1.49572829,
         0.9050209 , -1.73795667],
       [ 1.34868424, -0.2795409 ,  0.82598405,  1.23985085, -0.25282181,
         0.40371625, -0.75582549],
       [-1.51858249, -0.93623973, -1.91027753, -1.06667428, -1.58795674,
         0.11265797, -0.29680097],
       [-0.68893356,  0.63993546, -0.11717446, -0.23547223, -1.27142453,
         1.77913842, -0.28105363]])
```

In [28]:
```
np.random.randn((7,7))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-0701a3589c9f> in <module>
----> 1 np.random.randn((7,7))

mtrand.pyx in mtrand.RandomState.randn()

mtrand.pyx in mtrand.RandomState.standard_normal()

mtrand.pyx in mtrand.cont0_array()

TypeError: 'tuple' object cannot be interpreted as an integer
```

k. normal()

numpy.random.normal(loc=0.0, scale=1.0, size=None) Draw random samples from a normal (Gaussian) distribution.

Parameters :
loc : float -- Mean (“centre”) of the distribution. scale : float -- Standard deviation (spread or “width”) of the distribution. size : tuple of ints -- Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

In [29]:
```
np.random.normal()
```

Out[29]:
```
1.0980093126932042
```

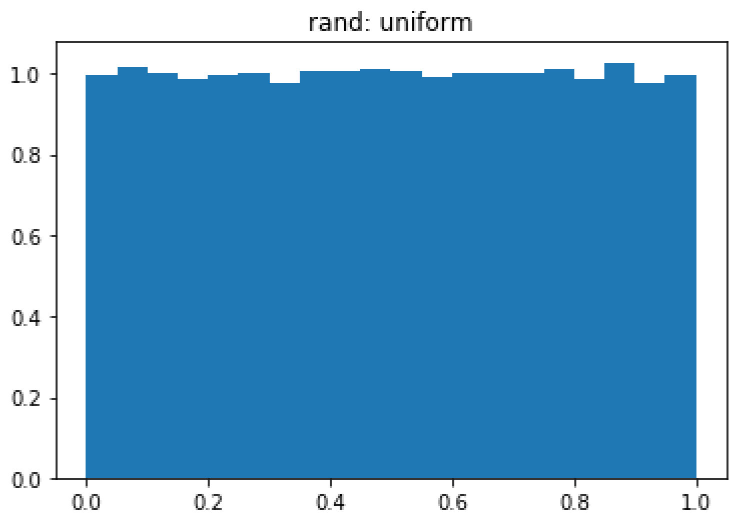
Difference between rand() and randn()

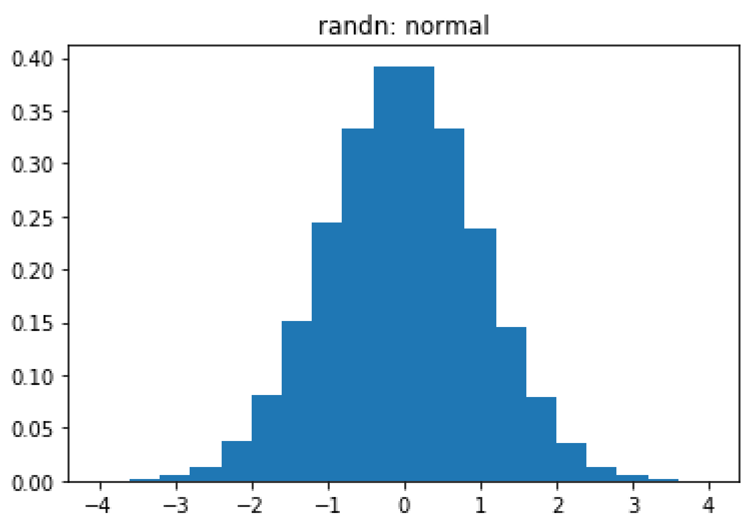
In [111]:

```
sample_size = 100000
uniform = np.random.rand(sample_size)
normal = np.random.randn(sample_size)

pdf, bins, patches = plt.hist(uniform, bins=20, range=(0, 1), density=True)
plt.title('rand: uniform')
plt.show()

pdf, bins, patches = plt.hist(normal, bins=20, range=(-4, 4), density=True)
plt.title('randn: normal')
plt.show()
```





L. randint()

Return random integers from `low` (inclusive) to `high` (exclusive).

Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [`low` , `high`). If `high` is None (the default), then results are from [0, `low`).

- Syntax : randint(low, high=None, size=None, dtype='l')

In [31]:
```
np.random.randint(1,100) #returns one random int, 1 inclusive, 100 exclusive
# np.random.randint(high=10,low=1,size=(2,3))
```

Out[31]: 73

In [32]:
```
np.random.randint(1,100,10) #returns ten random int,
```

Out[32]: array([15, 66, 91, 34, 83, 43, 64, 68, 15, 19])

m. choice() — By default replace = True ✓.

Docstring:

choice(a, size=None, replace=True, p=None)

np.random.choice(np.arrange(10),6)-- select 6 random numbers from the input array

In [33]:
```
y=np.random.choice(['a','b'],1000,p=[0.8,0.2])
np.unique(y, return_counts=True)
```

Out[33]: (array(['a', 'b'], dtype='<U1'), array([796, 204], dtype=int64))

n. reshape()

shapes an array without changing data of array.

- Syntax : reshape(array, shape, order = 'C')

In [34]:

```
# Python Program illustrating
# numpy.reshape() method

array = np.arange(8)
print("Original array : \n", array)

# shape array with 2 rows and 4 columns
array = np.arange(8).reshape(2, 4)
print("\narray reshaped with 2 rows and 4 columns : \n", array)

# shape array with 2 rows and 4 columns
array = np.arange(8).reshape(4 ,2)
print("\narray reshaped with 2 rows and 4 columns : \n", array)

# Constructs 3D array
array = np.arange(8).reshape(2, 2, 2)
print("\nOriginal array reshaped to 3D : \n", array)

```

```
Original array :
 [0 1 2 3 4 5 6 7]

array reshaped with 2 rows and 4 columns :
 [[0 1 2 3]
 [4 5 6 7]]

array reshaped with 2 rows and 4 columns :
 [[0 1]
 [2 3]
 [4 5]
 [6 7]]

Original array reshaped to 3D :
 [[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

Attributes of a NumPy :

- Ndim: displays the dimension of the array
- Shape: returns a tuple of integers indicating the size of the array
- Size: returns the total number of elements in the NumPy array
- Dtype: returns the type of elements in the array, i.e., int64, character
- Itemsize: returns the size in bytes of each item
- nbytes: which lists the total size (in bytes) of the array
- Reshape: Reshapes the NumPy array

In general, we expect that **nbytes** is equal to **itemsize times size**.

In [35]:
```python
np.random.seed(0)  # seed for reproducibility

x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

In [36]:
```python
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Each array has attributes **ndim** (the number of dimensions), **shape** (the size of each dimension), and **size** (the total size of the array):

In [37]:
```python
print("the type of elements in the array-dtype:", x3.dtype)
```

```
the type of elements in the array-dtype: int32
```

In [38]:
```python
print("itemsize of each item:", x3.itemsize, "bytes")
print("nbytes:total size (in bytes) of the array", x3.nbytes, "bytes")
```

```
itemsize of each item: 4 bytes
nbytes:total size (in bytes) of the array 240 bytes
```

Indexing & slicing of 1-D arrays (vectors)

Similar to the use of slice indexing with lists and strings, we can use slice indexing to pull out sub-regions of ndarrays.

In [39]:
```python
# Lets create a simple 1-D NumPy array.
# (we can use arange() as well.)
array_1d = np.array([-10, -2, 0, 2, 17, 106,200])
```

In [40]:
```python
array_1d
```

Out[40]:
```
array([-10,  -2,   0,   2,  17, 106, 200])
```

In [41]:
```python
# In the simplest case, selecting one or more elements of NumPy array looks
# Getting value at certain index
array_1d[0]
```

Out[41]:
```
-10
```

In [42]:
```python
# Getting a range value
array_1d[0:3], array_1d
# array_1d is included in the out to compare and understand
```

Out[42]:
```
(array([-10,  -2,   0]), array([-10,  -2,   0,   2,  17, 106, 200]))
```

In [43]:
```
# Using -ve index
array_1d[-2], array_1d
# array_1d is included in the out to compare and understand
```
Out[43]:
```
(106, array([-10,  -2,   0,   2,  17, 106, 200]))
```

In [44]:
```
# Using -ve index for  a range
array_1d[1:-2], array_1d # 1 inclusive and -2 exclusive in this case
```
Out[44]:
```
(array([-2,  0,  2, 17]), array([-10,  -2,   0,   2,  17, 106, 200]))
```

In [45]:
```
# Getting up-to and from certain index -- remember index starts from '0'
# (no need to give start and stop indexes)
array_1d[:2], array_1d[2:]
```
Out[45]:
```
(array([-10,  -2]), array([  0,   2,  17, 106, 200]))
```

In [46]:
```
# Assigning a new value to a certain index in the array
array_1d[0] = -102
```

In [47]:
```
array_1d
# The first element is changed to -102
```
Out[47]:
```
array([-102,   -2,    0,    2,   17,  106,  200])
```

To access any single element from 2D-Numpy, the general format is:

- **`array_2d[row][col]`**
  or
- **`array_2d[row,col]`** .

We will use `[row,col]` , easier to use comma ',' for clarity. However if we suppose to access the more than one element then these two expression will give different result.

In [48]:
```
an_array = np.array([[11,12,13,14], [21,22,23,24], [31,32,33,34]])
```

In [49]:
```
an_array
```
Out[49]:
```
array([[11, 12, 13, 14],
       [21, 22, 23, 24],
       [31, 32, 33, 34]])
```

In [50]:
```
an_array[:2, 1:3]
```
Out[50]:
```
array([[12, 13],
       [22, 23]])
```

In [51]:
```python
xx=an_array[:2]
xx
```

Out[51]:
```
array([[11, 12, 13, 14],
       [21, 22, 23, 24]])
```

In [52]:
```python
xx[1:3]
```

Out[52]:
```
array([[21, 22, 23, 24]])
```

In [53]:
```python
an_array[[1,2],[1,3]]
```

Out[53]:
```
array([22, 34])
```

In [54]:
```python
an_array[[0,1],[1,0]]
```

Out[54]:
```
array([12, 21])
```

In [55]:
```python
an_array[[0,1]][[1,0]]
```

Out[55]:
```
array([[21, 22, 23, 24],
       [11, 12, 13, 14]])
```

In [56]:
```python
# Rank 2 array of shape (3, 4)
an_array = np.array([[11,12,13,14], [21,22,23,24], [31,32,33,34]])
print(an_array)
```

```
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

Use array slicing to get a subarray consisting of the first 2 rows x 2 columns.

In [57]:
```python
a_slice = an_array[:2, 1:3]
print(a_slice)
```

```
[[12 13]
 [22 23]]
```

When you modify a slice, you actually modify the underlying array.

In [58]:
```python
print("Before:", an_array[0, 1])   #inspect the element at 0, 1
a_slice[0, 0] = 1000     # a_slice[0, 0] is the same piece of data as an_arra
print("After:", an_array[0, 1])
```

```
Before: 12
After: 1000
```

Use both integer indexing & slice indexing

We can use combinations of integer indexing and slice indexing to create different shaped matrices.

In [59]:

```python
import numpy as np
# Create a Rank 2 array of shape (3, 4)
an_array = np.array([[11,12,13,14], [21,22,23,24], [31,32,33,34]])
print(an_array)
```

```
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

In [60]:

```python
# Using both integer indexing & slicing generates an array of lower rank
row_rank1 = an_array[1, :]    # Rank 1 view

print(row_rank1, row_rank1.shape)  # notice only a single []
```

```
[21 22 23 24] (4,)
```

In [61]:

```python
# Slicing alone: generates an array of the same rank as the an_array
row_rank2 = an_array[1:2, :]  # Rank 2 view

print(row_rank2, row_rank2.shape)   # Notice the [[ ]]
```

```
[[21 22 23 24]] (1, 4)
```

In [62]:

```python
#We can do the same thing for columns of an array:

print()
col_rank1 = an_array[:, 1]
col_rank2 = an_array[:, 1:2]

print(col_rank1, col_rank1.shape)  # Rank 1
print()
print(col_rank2, col_rank2.shape)  # Rank 2
```

```

[12 22 32] (3,)

[[12]
 [22]
 [32]] (3, 1)
```

- **Array Indexing for changing elements:**
  - Sometimes it's useful to use an array of indexes to access or change elements.

In [63]:

```python
# Create a new array
an_array = np.array([[11,12,13], [21,22,23], [31,32,33], [41,42,43]])

print('Original Array:')
print(an_array)
```

```
Original Array:
[[11 12 13]
 [21 22 23]
 [31 32 33]
 [41 42 43]]
```

In [64]:

```python
# Create an array of indices
col_indices = np.array([0, 1, 2, 0])
print('\nCol indices picked : ', col_indices)

row_indices = np.arange(4)
print('\nRows indices picked : ', row_indices)
```

```

Col indices picked :  [0 1 2 0]

Rows indices picked :  [0 1 2 3]
```

In [65]:

```python
# Examine the pairings of row_indices and col_indices.  These are the elemen
for row,col in zip(row_indices,col_indices):
    print(row, ", ",col)
```

```
0 ,  0
1 ,  1
2 ,  2
3 ,  0
```

In [66]:

```python

# Select one element from each row
print('Values in the array at those indices: ',an_array[row_indices, col_ind
```

```
Values in the array at those indices:  [11 22 33 41]
```

In [67]:

```python
# Change one element from each row using the indices selected
an_array[row_indices, col_indices] += 100000

print('\nChanged Array:')
print(an_array)
```

```

Changed Array:
[[100011     12     13]
 [    21 100022     23]
 [    31     32 100033]
 [100041     42     43]]
```

Boolean Indexing / Fancy Indexing / Conditional Indexing

In [68]:
```
# create a 3x2 array
a = np.array([[1,2], [3, 4], [5, 6]])
a
```

Out[68]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [69]:
```
# create a filter which will be boolean values for whether each element meet
c=a > 2
print(c)
```

```
[[False False]
 [ True  True]
 [ True  True]]
```

Notice that the c is a same size ndarray as array a, array c is filled with True for each element whose corresponding element in array a is greater than 2 and False for those elements whose value is less than 2.

We can use , these comparison expressions directly for access. Result is only those elements for which the expression evaluates to True.

In [70]:
```
print(a[c])
print(a[c].shape)
```

```
[3 4 5 6]
(4,)
```

notice that the result is a 1D array.

Lets see if this works with writing mulitple conditions as well. In that process we'll also see that we dont have to store results in one variable and then pass for subsetting. We can instead, write the conditional expression directly for subsetting.

In [71]:
```
a>2
```

Out[71]:
```
array([[False, False],
       [ True,  True],
       [ True,  True]])
```

In [72]:
```
a<5
```

Out[72]:
```
array([[ True,  True],
       [ True,  True],
       [False, False]])
```

In [73]:
```
(a>2) | (a<5)
```
Out[73]:
```
array([[ True,  True],
       [ True,  True],
       [ True,  True]])
```

In [74]:
```
(a>2) & (a<5)
```
Out[74]:
```
array([[False, False],
       [ True,  True],
       [False, False]])
```

In [75]:
```
print(a)
print(a[(a>2) | (a<5)] )
a[(a>2) & (a<5)]  ###### A, B  i.e Multiple operation in one line
```
```
[[1 2]
 [3 4]
 [5 6]]
[1 2 3 4 5 6]
```
Out[75]:
```
array([3, 4])
```

Arithmetic Array Operations:

In [76]:
```
x = np.array([[111,112],[121,122]], dtype=np.int)
y = np.array([[211.1,212.1],[221.1,222.1]], dtype=np.float64)

print(x)
print()
print(y)
```
```
[[111 112]
 [121 122]]

[[211.1 212.1]
 [221.1 222.1]]
```

In [77]:
```
# add
print(x + y)          # The plus sign works
print()
print(np.add(x, y))  # so does the numpy function "add"
```
```
[[322.1 324.1]
 [342.1 344.1]]

[[322.1 324.1]
 [342.1 344.1]]
```

In [78]:
```
# subtract
print(x - y)
print()
print(np.subtract(x, y))
```

```
[[-100.1 -100.1]
 [-100.1 -100.1]]

[[-100.1 -100.1]
 [-100.1 -100.1]]
```

In [79]:
```
# multiply
print(x * y)
print()
print(np.multiply(x, y))
```

```
[[23432.1 23755.2]
 [26753.1 27096.2]]

[[23432.1 23755.2]
 [26753.1 27096.2]]
```

In [80]:
```
# divide
print(x / y)
print()
print(np.divide(x, y))
```

```
[[0.52581715 0.52805281]
 [0.54726368 0.54930212]]

[[0.52581715 0.52805281]
 [0.54726368 0.54930212]]
```

In [81]:
```
# square root
print(np.sqrt(x))
x
```

```
[[10.53565375 10.58300524]
 [11.         11.04536102]]
```

Out[81]:
```
array([[111, 112],
       [121, 122]])
```

In [82]:
```
# exponent (e ** x)
print(np.exp(x))
```

```
[[1.60948707e+48 4.37503945e+48]
 [3.54513118e+52 9.63666567e+52]]
```

In general you'll find that , mathematical functions from numpy [being referred as np here] when applied on array, give back result as an array where that function has been applied on individual elements. However the functions from package math on the other hand give error when applied to arrays. They only work for scalars.

In [83]:
```python
# square root
import math
math.sqrt(x)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-83-f63d9241fcd6> in <module>
      1 # square root
      2 import math
----> 3 math.sqrt(x)

TypeError: only size-1 arrays can be converted to Python scalars
```

np.dot() in Numpy

To multiply two matrices, dot () method is used. Here is an introduction to numpy.dot(a, b, out=None)

Few specifications of numpy.dot:

- If both a and b are 1-D (one dimensional) arrays — Inner product of two vectors (without a complex conjugation)
- If both a and b are 2-D (two dimensional) arrays — Matrix multiplication
- If either a or b is 0-D (also known as a scalar) — Multiply by using numpy.multiply(a, b) or a * b.
- If a is an N-D array and b is a 1-D array — Sum product over the last axis of a and b.
- If a is an N-D array and b is an M-D array provided that M>=2 — Sum product over the last axis of a and the second-to-last axis of b:

If the last dimension of a is not the same size as the second-to-last dimension of b.

In [84]:
```python
v = np.array([9,10])  #### Defining 1-D array
v
```

Out[84]:
```
array([ 9, 10])
```

In [85]:
```python
w = np.array([11, 12])  #### Defining 1-D array
w
```

Out[85]:
```
array([11, 12])
```

In [86]:
```python
# Matrix multiplication
v.dot(w)
```

Out[86]:
```
219
```

You can see that result is not what you'd expect from matrix multiplication. This happens because a single dimensional array is not a matrix.

In [87]:
```
print(v.shape)
print(w.shape)
```

```
(2,)
(2,)
```

In [88]:
```
v=v.reshape((1,2))
w=w.reshape((1,2))
v
```

Out[88]:
```
array([[ 9, 10]])
```

Now if you simply try to do v.dot(w) or np.dot(v,w) [both are same] , you will get and error because you can multiple a mtrix of shape 2X1 with a matrix of 2X1 .

In [89]:
```
np.dot(v,w)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-89-efb51945670c> in <module>
----> 1 np.dot(v,w)

ValueError: shapes (1,2) and (1,2) not aligned: 2 (dim 1) != 1 (dim 0)
```

In [90]:
```
print('matrix v : ',v)
print('matrix v Transpose:',v.T)
print('matrix w:',w)
print('matrix w Transpose:',w.T)
print('~~~~~~~~~ v multiply with transpose of w')
print(np.dot(v,w.T))
print('~~~~~~~~~ transpose of v is multiply by w')
print(np.dot(v.T,w))
```

```
matrix v :  [[ 9 10]]
matrix v Transpose: [[ 9]
 [10]]
matrix w: [[11 12]]
matrix w Transpose: [[11]
 [12]]
~~~~~~~~~ v multiply with transpose of w
[[219]]
~~~~~~~~~ transpose of v is multiply by w
[[ 99 108]
 [110 120]]
```

If you leave v to be a single dimensional array . you will simply get an element wise multiplication. Here is an example

In [91]:
```
print(x)
v=np.array([9,10])
print("~~~~~")
print(v)
x.dot(v)
```

```
[[111 112]
 [121 122]]
~~~~~
[ 9 10]
```

Out[91]:
```
array([2119, 2309])
```

In [92]:
```
print(x)
print("~~~")
print(y)
x.dot(y)
```

```
[[111 112]
 [121 122]]
~~~
[[211.1 212.1]
 [221.1 222.1]]
```

Out[92]:
```
array([[48195.3, 48418.3],
       [52517.3, 52760.3]])
```

Broadcasting

Numpy arrays are different from normal Python lists because of their ability to broadcast. We will only cover the basics, for further details on broadcasting rules, click here (https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html)
Another good read on broadcasting (https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html)!

Lets start with some simple examples:

In [93]:
```
# Lets create an array using arange()
array_1d = np.arange(0,10)
array_1d
```

Out[93]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Take a slice of the array and set it equal to some number, say 500.

```
array_1d[0:5] = 500
```

this will **broadcast the value of 500 to the first 5 elements** of the array_1d

In [94]:
```
array_1d[0:5] = 500
array_1d
```

Out[94]:
```
array([500, 500, 500, 500, 500,   5,   6,   7,   8,   9])
```

In [95]:
```python
# Lets create a 2D martix with ones
array_2d = np.ones((4,4))
array_2d
```

Out[95]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [96]:
```python
# Lets broadcast 300 to the first row of array_2d
array_2d[0] = 300
array_2d
```

Out[96]:
```
array([[300., 300., 300., 300.],
       [  1.,   1.,   1.,   1.],
       [  1.,   1.,   1.,   1.],
       [  1.,   1.,   1.,   1.]])
```

In [97]:
```python
# Lets create a simple 1-D array and broadcast to array_2d
array_2d + np.arange(0,4)
# try array_2d + np.arange(0,3), did this work? if not why?
```

Out[97]:
```
array([[300., 301., 302., 303.],
       [  1.,   2.,   3.,   4.],
       [  1.,   2.,   3.,   4.],
       [  1.,   2.,   3.,   4.]])
```

In [98]:
```python
array_2d + np.arange(0,4)
```

Out[98]:
```
array([[300., 301., 302., 303.],
       [  1.,   2.,   3.,   4.],
       [  1.,   2.,   3.,   4.],
       [  1.,   2.,   3.,   4.]])
```

In [99]:
```python
np.arange(0,4).shape
```

Out[99]:
```
(4,)
```

In [100]:
```python
array_2d
```

Out[100]:
```
array([[300., 300., 300., 300.],
       [  1.,   1.,   1.,   1.],
       [  1.,   1.,   1.,   1.],
       [  1.,   1.,   1.,   1.]])
```

In [101]:
```python
array_2d + 300
# array_2d + [300,2], did it work? if not why?
```

Out[101]:
```
array([[600., 600., 600., 600.],
       [301., 301., 301., 301.],
       [301., 301., 301., 301.],
       [301., 301., 301., 301.]])
```

In [102]:

```python
# array_2d + [300,2]
array_2d + [300,2,4,5]
```

Out[102]:

```
array([[600., 302., 304., 305.],
       [301.,   3.,   5.,   6.],
       [301.,   3.,   5.,   6.],
       [301.,   3.,   5.,   6.]])
```

Another broadcasting example

In [103]:

```python
array_1 = np.arange(1,4)
array_2 = np.arange(1,4)[:, np.newaxis]
```

In [104]:

```python
array_1
```

Out[104]:

```
array([1, 2, 3])
```

In [105]:

```python
array_2
```

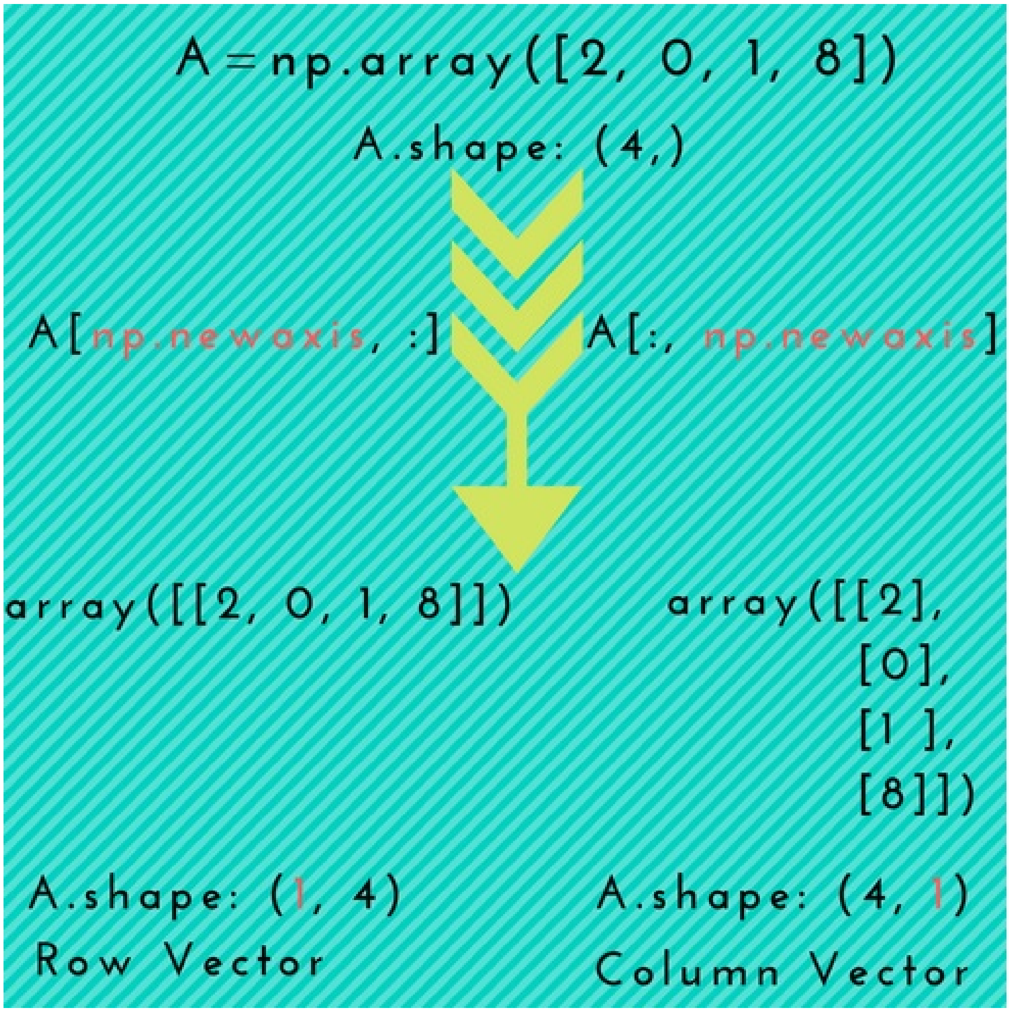
Out[105]:

```
array([[1],
       [2],
       [3]])
```

In [106]:

```python
from IPython.display import Image
Image("newaxis.jpg")
```

Out[106]:



In [107]:

```python
# Official way of printing is used, format() and len() are used for revision
print(array_1)
print("Shape of the array is: {}, this is {}-D array".format(array_1.shape,l
# (3,) indicates that this is a one dimensional array (vector)
```

```
[1 2 3]
Shape of the array is: (3,), this is 1-D array
```

In [108]:

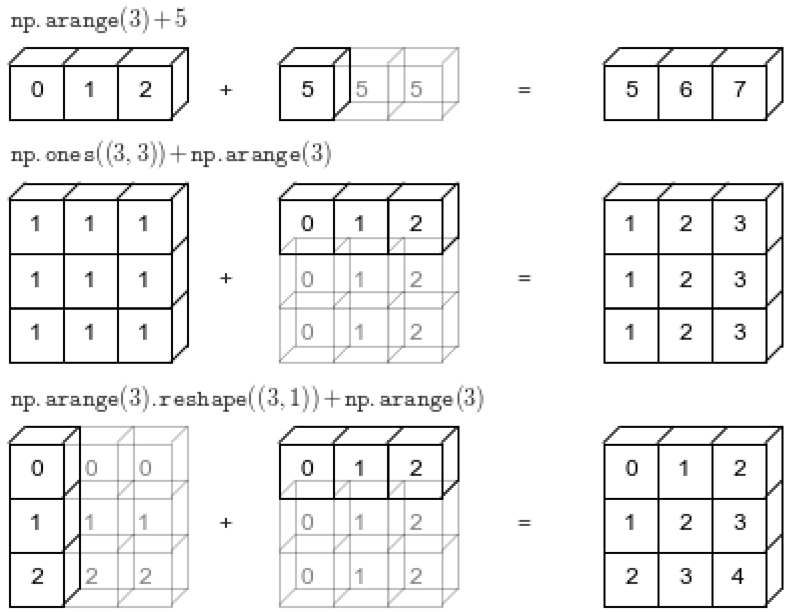
```python
# Official way of printing is used, format() and len() are used for revision
print(array_2)
print("Shape of the array is: {}, this is {}-D array".format(array_2.shape,l
# (3, 1) indicates that this is a 2-D array (matrix)
```

```
[[1]
 [2]
 [3]]
Shape of the array is: (3, 1), this is 2-D array
```

In [109]:
```
# Broadcasting arrays
array_1 + array_2
```

Out[109]:
```
array([[2, 3, 4],
       [3, 4, 5],
       [4, 5, 6]])
```

This image (https://jakevdp.github.io/PythonDataScienceHandbook/figures/02.05-broadcasting.png) could be very helpful to understand the broadcasting concepts: The code to generate this image is available here (https://jakevdp.github.io/PythonDataScienceHandbook/06.00-figure-code.html#Broadcasting).



Some other useful functions in Numpy

1. unique()

   y = np.random.choice(['a','b'], 1000, p=[0.8, 0.2])

   eg unique(y, return_counts = True)

   → give how many a's + how many b's.

2. **sort()** → sort the numpy array in ascending order. It sort the value "Inplace".

   eg x.sort()

3. **argsort()** → It gives the index order as per ascending sorting.

   eg x = [50, 25, 5, 95, 58]
   (index 0, 1, 2, 3, 4)

   x.argsort() $\xrightarrow{O/P}$ [2, 1, 0 3, 4] → It also a numpy array.

4. **max()** :- It gives maximum, value in numpy array.

5. argmax() :- corresponding index of maximum value.

6. sum() :- parameter; axis = None 'by default' ie sum() will give the sum of all values.

   if axis = 0 (R0, R1) Row movement ie accross list | axis = 1 along the list (C0, C1)